Programming Modular Robots with the TOTA Middleware

Marco Mamei and Franco Zambonelli

Dipartimento di Scienze e Metodi dell'Ingegneria,
University of Modena and Reggio Emilia
Via Allegri 13, 42100 Reggio Emilia, Italy
{mamei.marco, franco.zambonelli}@unimore.it

Abstract. Modular robots represent a perfect application scenario for multiagent coordination. The autonomous modules composing the robot must coordinate their respective activities to enforce a specific global shape or a coherent motion gait. Here we show how the TOTA ("Tuples On The Air") middleware can be effectively exploited to support agents' coordination in this context. The key idea in TOTA is to rely on spatially distributed tuples, spread across the robot, to guide the agents' activities in moving and reshaping the robot. Three simulated examples are presented to support our claims.

1 Introduction

A modular (or self-reconfigurable) robot is a flexible robot made up of a collection of (typically simple) autonomous elements connected with each other with some degree of freedom in their relative movements [1–6].

The key idea underlying a modular robot is to have its components execute distributed control algorithms so as to coordinate their actions and let the robot assume a specific shape or move according to some specific motion pattern (i.e., gait).

The flexibility of modular robots is highly desirable for tasks to be performed in hostile environments, such as fire fighting, search and rescue after an earthquake, and battlefield reconnaissance [7–10]. In these cases, robots can encounter unexpected situations and obstacles, hard to overcome for fixed-shape monolithic robots. A modular robot, instead, could shape itself depending on needs. For example, to pass through a hole, the robot can transform itself into a sort of snake; to move through a downhill slope, it can assume a circular shape and roll; to enter a room through a closed door, a modular robot may disassemble itself into a set of smaller units, crawl under the door, and then reassemble itself in the room.

Modular robots represent a perfect application scenario for multiagent systems. On the one hand, the modules constituting the modular robot should be autonomous and running agent applications. This, in fact, avoids single point of failure and bottlenecks. If one module breaks down the other can reorganize

their activities leaving the broken element behind. Autonomy enables also disconnected operations: modules can disassemble, move along different paths and reassemble subsequently. On the other hand, the main task the modules have to undertake is *coordination*. The agents governing the modules must coordinate their respective activities to enforce a specific global shape or a coherent motion gait in the whole robot (i.e., multiagent system).

For the purpose of supporting agent coordination, in our research, we developed a general middleware called "Tuples On The Air" (TOTA). TOTA supports the creation of distributed overlay data structures spread across a distributed network. Up to now, TOTA has been proved useful in developing a number of multiagent applications in distributed computing systems including: motion coordination in pervasive computing scenarios, on-demand routing protocols in MANET, P2P protocols and swarm intelligence algorithms [11, 12]. Implementing all these applications becomes relatively simple when suitable overlay data structures can be spread across the network to guide agents' coordination activities.

In this paper, we want to further test the TOTA's generality by implementing some advanced mechanisms required for the control of a modular robot. In particular, our goal is to show that TOTA enables to easily and flexibly express and program biologically-inspired control algorithms (i.e. hormone-based [5]) for modular robots' coordination.

The rest of the paper is organized as follows. Section 2 briefly presents the main concepts at the core of the TOTA middleware and its programming model. Section 3 focuses on a specific hormone-based approach to control modular robots (as adopted by [5]) and shows how it can be implemented by means of the TOTA middleware. Moreover, we explain the advantages of our implementation and discuss some related works. Section 4 presents three examples of modular robot's coordination using TOTA. The examples have been implemented on the Polybot modular robot simulator [4]. Finally, Section 5 presents some concluding remarks.

2 The Tuples on the Air Approach

TOTA is a general-purpose middleware for multiagent coordination, in distributed computing scenarios [12]. In TOTA we assume the presence of a network of possibly mobile nodes, each running an agent application. Each agent is supported by a local version of the TOTA middleware. Nodes are connected by only short-range network links; there are not long backbones in the network. Each agent has only a local (one-hop) perception of its environment. Long range interactions must be mediated by other agents. Upon the distributed space identified by the dynamic network of TOTA nodes, each agent is capable of locally storing tuples [13] and letting them diffuse through the network. Tuples are injected in the system from a particular node, and spread hop-by-hop accordingly to a specified propagation rule (see Figure 1(top)). In the modular robot scenario, this reflects in having each agent installed in a module of the robot. The

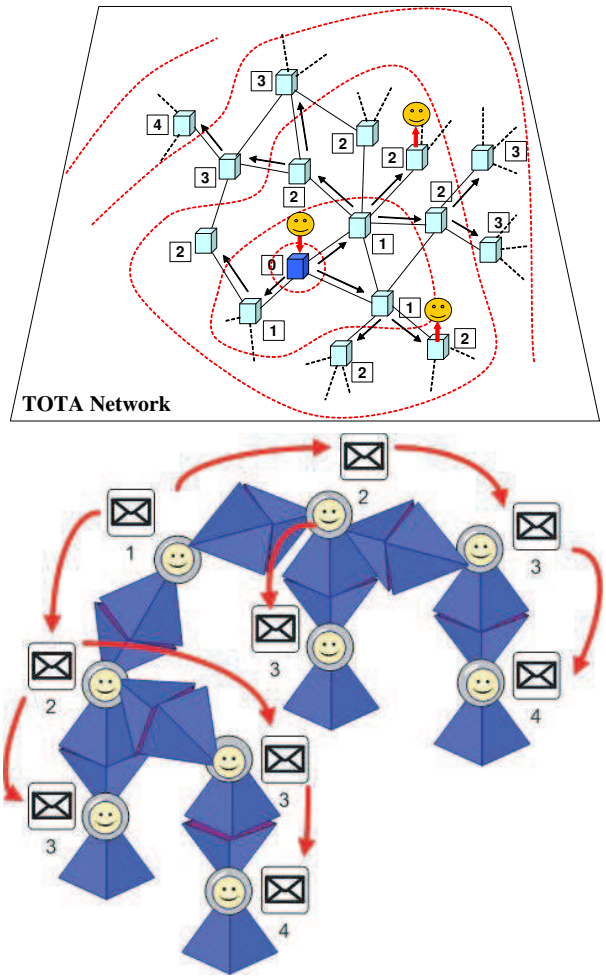

Fig. 1. (top) The general scenario of TOTA: application components live in an environment in which they can inject autonomously propagating tuples and sense tuples present in their local neighborhood. The environment is realized by means of a peer-to-peer network in which tuples propagates by means of a multi-hop mechanism. (bottom) TOTA in modular robots.

modules (and thus the agents) are connected according to the topology of the robot. Tuples are injected and propagate across the robot's body (see Figure 1(bottom)).

In TOTA, distributed tuples **T=(C,P,M)** are characterized by a content **C**, a propagation rule **P** and a maintenance rule **M**. The content **C** is an ordered set of typed fields representing the information carried on by the tuple. The propagation rule **P** determines how the tuple should be distributed and propagated in the network. This includes determining the "scope" of the tuple (i.e. the distance at which such tuple should be propagated and possibly the spatial direction of propagation) and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rules can determine how tuple content should change while it is propagated to actually create distributed data structures. The maintenance rule **M** determines how a tuple's distributed structure should react to events occurring in the environment. For instance, when new nodes get in touch with a network, TOTA automatically checks the propagation rules of the already stored tuples and eventually propagates the tuples to the new nodes. Similarly, when the topology changes due to nodes' movements, the distributed tuple structure changes to reflect the new topology.

From the application components' point of view, executing and interacting basically reduces to define and inject tuples in the network (*inject* method) and to read local (*read* method) and one-hop neighbor (*readOneHop* method) tuples via a pattern-matching mechanism. TOTA provides a compact API to perform these operations.

We developed a first prototype of TOTA running on Linux IPAQs equipped with 802.11b WLAN and Java (J2ME, CDC, Personal profile). Moreover, we have implemented an emulator to analyze TOTA behavior in presence of hundreds of nodes [11]. To perform experiments with modular robots, we connected our TOTA simulator with the Polybot modular robot simulator [4]. This program can simulate the behavior of various types of modular robots, taking into account both the characteristics of the joints connecting different parts of the robot and the physical forces acting on the robot (e.g., gravitation), and offers a 3D view of the robot's actual configuration and movements. Distributed algorithms to control the robot can be implemented with this simulator. Specifically, each module of the robot is provided with an API enabling us to drive the module actuator, and to sample and possibly change the way in which the robot is connected to the other modules. To connect this simulator with the TOTA one, we created an object having access both to the TOTA API and to the modular robot API. This object "runs," at the same time, in the TOTA simulator and in the modular robot simulator, connecting the two.

3 Multiagent Coordination in Modular Robots

From a multiagent perspective, the main challenge in modular robots is the design of decentralized coordination mechanisms enabling autonomous agents (i.e.,

modules) to coordinate their actions to achieve a specific global shape, or a specific motion gait. Some of the most innovative approaches to control a modular robot adopt the biologically inspired idea of hormones. These approaches have been used to control the CONRO modular robot and directly inspired our work [5]. Hormone signals are actually sort of messages, spread across the robot and triggering the individual actuator's bending. For example, a "head" module in a modular robot (see later for details) can inject in the robot a sequence of hormone signals. All the other modules can be programmed to react to the income of such signals by bending their actuator by a specified angle. A motion gait would be encoded by means of a specific sequence of hormones to be injected in the robot and by means of specific reactions triggered by these hormones, changing the bending angles.

The idea of hormones is a perfect match for TOTA distributed tuples and our approach has been to re-implement hormones with the support of our middleware. With this regard, it is fair to remark that we do not propose a novel approach for modular robot coordination. We just take advantage of the TOTA middleware to implement (with some changes) the hormone-based approach [5]. In particular, the main subject of our research has been the chain-type modular robot. This kind of robots is characterized by the fact modules are connected in a line configuration (e.g., snake-like), or - eventually - in a tree-like configuration (e.g., robot with legs), see the figures in next pages. In our experiments, we assumed that the robot is composed of very simple equal modules (i.e., joint actuators). Each module has a "front" side and a "back" side. On each side there are two docking points and an infrared network link. The two docking points enable a module to physically connect with other ones. This is of course fundamental to actually building the chain constituting the modular robot. The infrared (IR) link enables communication between connected modules (see Figure 3). Modules connect by their IR links in a network, resembling the robot topology.

Each module runs the TOTA middleware and an agent in charge of driving the module joint. The agent, looking at the active IR links, is able to infer whether it is the "head," the "tail," or a part of the "body" of the robot. Specifically, the "head" agent is the one with only the back IR link active, the "tail" agent is the one with only the front IR link active, a "body" agent is one having both the IR links active. The process of assessing whether an IR link is active or not can be based on "ping" messages and can be executed iteratively to take into account topological reconfigurations and module breakdown.

From a methodology point of view and in very general terms, the proposed approach consists in codifying a motion gait by means of a *Gait* tuple. Such a tuple has the structure depicted in Figure 2. Once this tuple is injected in the modular robot, it propagates hop by hop across all the modules changing its content to the desired gait. For the upcoming discussion – where we will present concrete code samples – it is important to remark that the, in TOTA, tuples are implemented by means of objects. The TOTA middleware executes the tuple's methods to "animate" it. In particular the *changeTupleContent* method allows

a tuple to change its content and the *move* method actually propagate the tuple to neighbor nodes where it will be executed again.

Abstract GaitTuple
C = (id, angle)
P = (propagate hop-by-hop, changing the content so as to encode in the "angle"-distributed data structure the shape the robot has to assume)
M = (if the network topology changes, restart propagation according to new head, body and tail position)

Fig. 2. Structure of the abstract *GaitTuple*. This tuple encodes in its distributed shape (i.e., angle field values) the form we want the robot to assume

When an agent installed on a module senses the income of a tuple of this kind, it reacts by bending the module joint by the angle specified in the tuple. So, for example, if in a robot composed of N modules a tuple having in its content a fixed angle of about $(360/N)^\circ$ is spread, the robot closes into a loop. More specifically, the presented approach (that is strongly inspired by [5]) is based on the following key points.

1. The head agent injects in the network (i.e., in the robot modules) a specific GaitTuple, representing the shape (or a step of the gait) the robot has to assume.
2. The tuple propagates from the head to the tail letting the robot bend accordingly.
3. When the tail receives the tuple, it injects another tuple (with constant value) for the purpose of notifying the head that the Gait Tuple completed its travel.
4. When the head receives the above constant tuple it can inject another *GaitTuple* implementing the second configuration the robot has to assume (i.e., second step in a motion gait). Or, alternatively, the constant tuple can automatically trigger a change in the content of the *GaitTuple* to let the robot assume the second configuration.
5. The process continues iteratively.

3.1 Related Approaches

As already introduced, the research on CONRO modular robot [5] directly inspired our experiments. However, the control mechanism, based on TOTA, extends the original hormone-based approach. TOTA tuples are active data structures and can change while being stored in the modular robot. Thus, even a complex gait can be obtained by using just one TOTA tuple that changes to

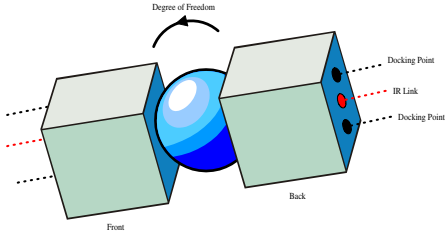

Fig. 3. A single module composing the modular robot. In our research we focus on simple module having just two docking points and two IR network links one for each side (front and back) of the module.

let the robot assume different configurations. On the contrary original hormones are passive and modules have to use several hormones to create a single gait.

In another research [2], each module of the modular robot runs a simple finite state automaton in which state transitions are driven by the local state, the state of neighbor modules, their locations, and some external information. Communications are limited to the immediate neighborhood and a limited number of bits are exchanged at each time step. The goal is not to create an exact predefined shape, but a structure with the correct properties (structural, morphological, etc.). Any stable "emergent" structure that exhibits the desired properties is considered satisfactory, with no regard for the "optimality" or details of the resulting geometry. This approach is very similar to ours, although our goal is to actually create engineered shapes and gaits, and not just purely emergent ones. The works [14, 3] goes further and introduces a compiler to automatically derive local rules from an high-level shape description.

The research in [15, 6] presents an interesting approach oriented toward self-reconfiguration and shape formation. In this approach, the desired configuration is grown from an initial seed agent. The initial seed produces growth by creating a gradient (similar to a TOTA distributed tuple), which attracts other agents. Once positioned, the agents become seed to let the shape grow again. The growth is guided by a representation of the desired configuration, which can be automatically generated from a 3D CAD model. Although very powerful, this approach focuses on shape formation and almost disregards gait coordination. Moreover, the approach assumes the presence of individual modules, more complex than the ones presented here.

Another thread of research, in modular robots, involves conceptually centralized control mechanisms [16]. In these approaches, a control table, specifying how each module must bend its actuator, is compiled off-line and then uploaded into the modules. The main advantage of this approach is that it allows us to design even complex motion gaits rather easily. The main drawback is that the control table is built for a specific robot configuration, and if the robot changes (e.g., new modules get connected), the table must be rebuilt from scratch. The research

in this area is mainly oriented to devising new languages to build the control table. One of the most advanced proposals is PARSL (Phase Automata Robot Scripting Language) [4]. PARSL is a scripting language based on XML syntax, designed to express motion gaits for chain-type modular robots. In PARSL it is possible to design a motion gait by means of abstract "waves of activity" traveling across the robot. Such high-level description is then automatically compiled to create the control table.

4 Experiments

In the rest of this section we will use TOTA to support three motion gaits in modular robot: the "caterpillar gait" (that lets the robot proceed by mimicking the motion of a snake) and the "rolling gait" (that lets the robot close in a loop, and then roll). Finally, we will present a gait for legged robots where agents coordinate legs' movements to proceed forward. It is worth noting that while the mechanisms underlying the former two gaits are well known in modular robot research [5, 16], the latter one has been designed from scratch.

4.1 Caterpillar Gait

To implement the caterpillar gait, the head agent starts the movement by injecting a caterpillar tuple (i.e., a TOTA tuple of the class *CaterpillarGaitTuple*). The general structure of such a tuple is depicted in Figure 4, it propagates across the robot letting it bend accordingly. Once the tail agent receives the tuple, according to the general description given above, it injects another tuple to notify the head that a new step is ready to be executed. At this point, the head agent updates the *Caterpillar GaitTuple* accordingly to the Table 5 and injects it again. Once spread, this tuple lets the gait proceed by another step. Useful insights to understand how the caterpillar gait works and how the Table 5 has been compiled can be found in Figure 6. The code implementing the *CaterpillarGaitTuple* tuple can be found in Figure 7. This process is iterated letting the whole robot move performing the caterpillar gait (see Figure 8).

CaterpillarGait Tuple
C = (state, angle)
P = (propagate hop-by-hop, storing on intermediate nodes changing the content accordingly to the table in Figure 5)
M = (If on the head node and upon the receipt of a gait-tuple, re-apply propagation)

Fig. 4. The structure of the *CaterpillarGaitTuple* tuple

Current State	New State	New Angle
INIT	A	$+45°$
A	B	$+45°$
B	C	$-45°$
C	D	$-45°$
D	A	$+45°$

Fig. 5. This table shows how the content of the *CaterpillarGaitTuple* changes

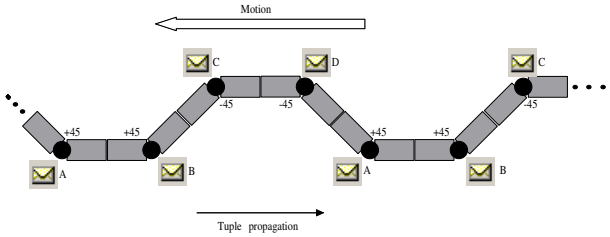

Fig. 6. The caterpillar gait works by letting a pattern of activity travel along the robot, letting it going forward

```
public class CaterpillarGaitTuple extends GaitTuple {
 /* constant declaration as in caterpillar gait table */
 /* tuple sates: INIT, A,B,C,D and respective angles
 degA,degB,degC,degD are defined */

 public int state = INIT;
 public int angle = 0;

 protected void changeTupleContent()
 {
  switch(state)
  {
   case INIT : state = A;
               angle = degA;
               break;
   case A    : state = B;
               angle = degB;
               break;
   case B    : state = C;
               angle = degC;
               break;
   case C    : state = D;
               angle = degD;
               break;
   case D    : state = A;
               angle = degA;
               break;
  }
 }
}
```

Fig. 7. The code realizing the *CaterpillarGaitTuple* TOTA class

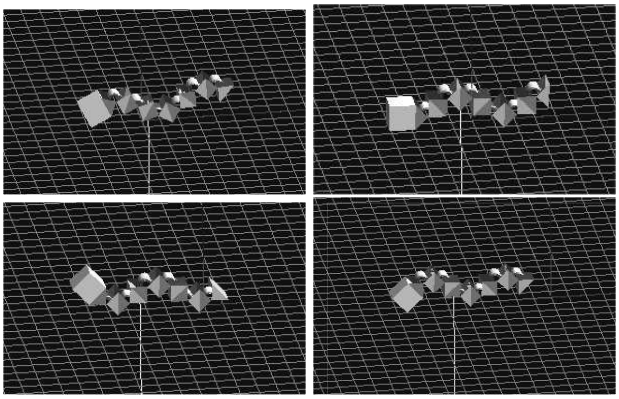

Fig. 8. Some stages of a caterpillar gait, in a chain-typed modular robot, composed of six actuators

4.2 Rolling Gait

The idea of this gait is to let the robot close in a loop and then roll. Unfortunately, the modular robot simulator we employed does not allow structures with loops. Structures with loops are overconstrained. The simulation does not solve the constraint satisfaction problem. The simulation does not detect self-collision either. To overcome this problem, we let the robot bend in an open loop (something like a 'C' shape) and then roll. Although this complicates the rolling procedure, it allows us to maintain the general approach described before. In fact, we still have a "head" and a "tail" agent that would be otherwise removed if the loop were actually closed (i.e., with only "body" agents). The *RollingGait-Tuple* is the tuple employed to let the robot roll. In general terms, it can have two states, T (turn) and F (flat). Consider, for example, a robot composed of 12 modules and assuming a turning angle of 45°. A tuple spread in the robot with a distributed shape like "FTTFTTFTTFTT" (see Figure 9) closes in a loop. Then, if the tuple changes its content by "rolling" the above string (like the ROL assembler command), the robot performs the rolling gait. From the single tuple point of view, this consists in changing its content to assume values F - T - T iteratively. It is worth noting that such a kind of content change critically depends on the number of modules composing the robot and the number of turns we want to implement to let the robot close in a loop. For example, three turns of 60° each create a triangular track, four turns of 90° each create a rectangular track, etc. Moreover, it depends on the number of modules involved in each turn. For example, in Figure 9, two modules bend by 45° to create a 90° turn. Despite all these parameters, it is rather easy to build a general algorithm enabling a tuple to create dynamically, at runtime, the sequence of F and T states it has to cycle (e.g., F - T - T) to enable the rolling gait. A general description of the *RollingGaitTuple* enabling the rolling gait in the case of a robot composed of 12

modules and assuming four turns of 90° each, split between two modules bending by 45°, is illustrated in Figure 10. The code realizing the *RollingGaitTuple* can be found in Figure 11. Some snapshots showing the rolling gait in action are in Figure 12.

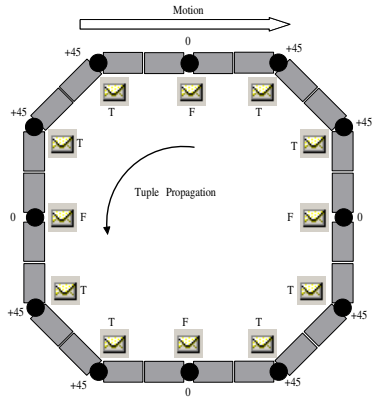

Fig. 9. In the rolling gait, the robot moves in one direction by shifting the turning modules (T) to the opposite direction

Rolling Gait Tuple
C = (state, angle)
P = (propagate hop-by-hop, cycling between the states F - T - T. Set the angle to 45deg if the state is T. Set the angle to 0 if the state is F)
M = (if the network topology changes, restart propagation according to new head, body and tail position)

Fig. 10. The rolling gait tuple

4.3 Walking Gait

We performed this experiment on a "legged" robot that can move by coordinating legs movements. The idea of this experiment is to have the modules of the robot connected in a 6-legs configuration (see Figure 13), and then coordinate the actions of these modules so as to let the legged robot walk. In this example, the robot is built from two types of modules (both available from the Polybot simulator): joints and connectors. Among the several possible configurations

```
public class RollingGaitTuple extends GaitTuple {
 // number of modules composing the robot
 private static final int N_MODULES;
 // number of turning points
 private static final int N_TURNS;
 // radius of the turn
 private static final int RADIUS ;
 // turning angle in deg
 private static final int TURN;

 public int state = 0;
 public int angle = 0;

 protected void changeTupleContent()
 {
  if(this.getSourceFromId().equals(tota.toString()))
   state = (state + 1)% N_MODULES;

  int mod = Integer.parseInt(tota.toString().substring(1));

  boolean cond = false;
  for(int i=0;i<N_TURNS;i++)
  {
   boolean cond1 =
   ((state+(i*N_MODULES)/N_TURNS) % N_MODULES) == mod;
   boolean cond2 =
   ((state+(i*N_MODULES)/N_TURNS) % N_MODULES) ==
   ((mod + RADIUS)% N_MODULES);
   if(cond1 || cond2)
   {
    cond = true;
    break;
   }
  }

  if(cond)
   angle = TURN;
  else
   angle = 0;
 }
}
```

Fig. 11. The code realizing the *RollingGaitTuple* TOTA class

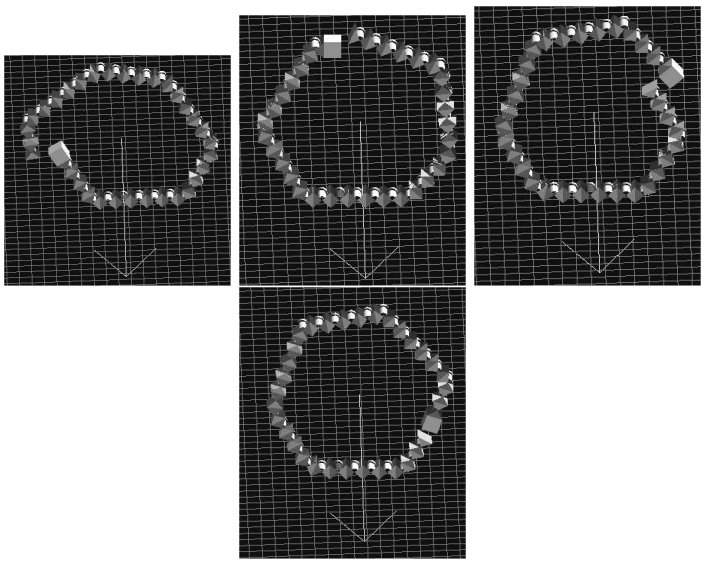

Fig. 12. Some stages of a rolling gait in a chain-typed modular robot composed of 32 actuators.

upon which it is possible to build a legged robot, the one we choose presents three key advantages:

- The adopted configuration is very modular. In order to crate a robot with more legs it is sufficient to add other legs (in multiples of two) at the end of the previous robot.
- The robot is highly flexible. It can swing in pitch and yaw both the backbone and the legs.
- Most importantly for the upcoming discussion, modules have a direction (front-rear) and they can distinguish both the kind of module to which the are attached (i.e., joint or connector) and the orientation of the connection (i.e., pitch-pitch, yaw-yaw, or pitch-yaw), see Figure 14(left). Thus each module can infer its position within the robot. Since connectors have no degrees of freedom – they are passive components – they do not need to localize. In particular it is possible to identify the following six important *roles* for modules: HEAD, SPINE, LEFT-SHOULDER, RIGHT-SHOULDER, LEFT-LEG, and RIGHT-LEG (see Figure 14(right)).

The robot in Figure 13 is in the rest mode. The first tuple we envisioned is the one forcing the robot to stand up (see Figure 15). The code of this *StandUpTuple* tuple (reported in Figure 16) is really simple. It basically forces all leg modules to turn 90°. More precisely, the way in which modules are connected implies that left legs should bend by 90°, while right legs by −90°. Once the robot is standing up, it can start moving the legs to proceed upward. This again is realized by

letting the head of the robot inject another tuple that propagates across the modules. This *WalkerGaitTuple* tuple is very simple: it alternatively lets the left and right robot legs swing 45° forward. The code of this tuple is reported in Figure 17, while some screen-shots of the actual robot movement are presented in Figure 18.

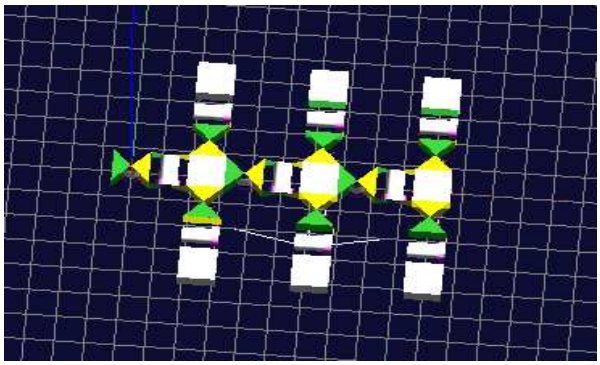

Fig. 13. A modular robot arranged in a 6-legged configuration

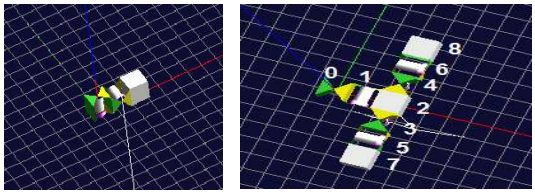

Fig. 14. (left) Detail of a robot leg. The pitch-yaw orientation between the two modules allows high flexibility. (right) Different neighbor connections allow each module to estimate its role within the robot (0 = HEAD, 1 = SPINE, 3 = LEFT-SHOULDER, 4 = RIGHT-SHOULDER, 5 = LEFT-LEG, 6 = RIGHT-LEG)

5 Conclusions

In this paper we applied the TOTA middleware to support agents' coordination in a modular robot application scenario. As illustrated with concrete code examples, TOTA distributed tuples represent a valid abstraction to guide agents' activities. Despite the simplicity of the experiments reported in this paper, we

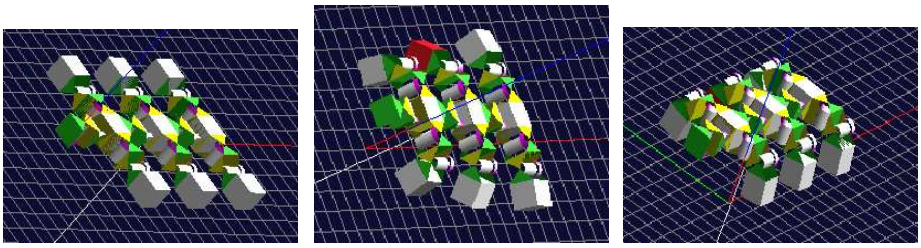

Fig. 15. A 6-legged robot stands up

```
public class StandUpTuple extends GaitTuple {

 // bend degree
 private static final int Deg = 90;
 public int angle;

 protected void changeTupleContent()
 {
  int role = (RoleTuple)tota.read(new RoleTuple()).role;
  if (role == RIGHT-LEG) angle = -Deg;
  else if (role == LEFT-LEG) angle = Deg;
  else angle = 0;
 }
}
```

Fig. 16. The code realizing the *StandUpTuple* TOTA class.

```
public class WalkerGaitTuple extends GaitTuple {
 //states
 private static final int FORWARD = 0;
 private static final int REVERSE = 1;

 public int state = FORWARD;
 public int angle = 0;

 protected void changeTupleContent()
 {
  int role = (RoleTuple)tota.read(new RoleTuple()).role;

  if (role == RIGHT-LEG) angle = -45;
  if (role == LEFT-LEG) angle = 45;
  if (role == SPINE) angle = 0;

   if (state == FORWARD)
   {
    if (role == LEFT-SHOULDER) angle = 0;
    if (role == RIGHT-SHOULDER) angle = -45;
    state = REVERSE;
   }
   else
   {
    if (role == LEFT-SHOULDER) angle = 45;
    if (role == RIGHT-SHOULDER) angle = 0;
    state = FORWARD;
   }
 }
```

Fig. 17. The code realizing the *WalkerGaitTuple* TOTA class.

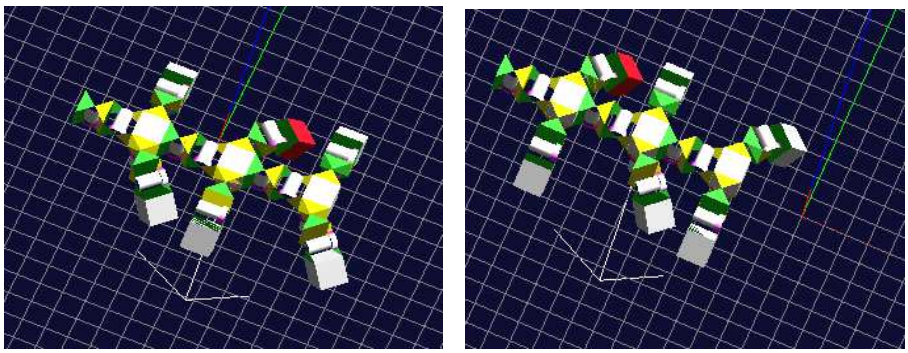

Fig. 18. The legged robot walks by coordinating legs movements.

are confident that the capability of distributed tuples (and of TOTA middleware) will have an important role in future research. However, we are also aware that the widespread exploitation of the proposed mechanisms will require the identification of proper methodologies to help designers in the development of complex applications (e.g. complex shape or articulated motion gait). In addition to that, we must also recognize that additional mechanisms – not dealt with by this paper – may also have an important role in this field. These include game-theoretic approaches [17] and cellular automata approaches [18, 19]. Part of our future work is about studying how these other approaches can be integrated with ours.

6 Acknowledgments

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

References

1. Balch, T., Parker, L.: Robot Teams: From Diversity to Polymorphism. A K Peters (2002)
2. Bojinov, H., Casal, A., Hogg, T.: Emergent structures in modular self-reconfigurable robots. In: Proceedings of the Intlernational Conference on Robotics and Automation. IEEE CS Press, San Francisco, California, USA (2000)
3. Jones, C., Mataric, M.: From local to global behavior in intelligent self-assembly. In: Proceedings of the Conference on Robotics and Automation. IEEE Press, Taipei, Taiwan (2003)
4. : (Modular reconfigurable robotics at parc) http://www2.parc.com/spl/projects/modrobots.
5. Shen, W., Salemi, B., Will, P.: Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots. IEEE Transactions on Robotics and Automation **18** (2002) 1 – 12
6. Stoy, K., Nagpal, R.: Self-reconfiguration using directed growth. In: 7th International Symposium on Distributed Autonomous Robotic Systems. Springer-Verlag, Toulouse, France (2004)
7. Noda, I., Takahashi, T., Morita, S., Koto, T., Tadokoro, S.: Language design for rescue agents. In Tanabe, M., van den Besselaar, P., Ishida, T., eds.: Digital Cities II. Springer (2002) 371–383
8. Rybski, P., Stoeter, S., Papanikolopoulos, N., Burt, I., Dahlin, T., Gini, M., Hougen, D.F., Krantz, D.G., Nageotte, F.: Sharing control: Presenting a framework for the operation and coordination of multiple miniature robots. Robotics and Automation Magazine **9** (2002) 41 – 48
9. Svennebring, J., Koenig, S.: Building terrain-covering ant robots. Autonomous Robots **16** (2004) 313 – 332
10. Tambe, M., Bowring, E., Jung, H., Kaminka, G., Maheswaran, R., Marecki, J., Modi, P., Nair, R., Okamoto, S., Pearce, J., Paruchuri, P., Pynadath, D., Scerri, P., Schurr, N., Varakantham, P.: Conflicts in teamwork: Hybrids to the rescue. In: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems. ACM Press, Utrecht, Netherlands (2005) 415 – 422

11. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the tota middleware. In: Proceedings of the International Conference On Pervasive Computing (Percom). IEEE CS Press, Orlando, Florida, USA (2004)
12. Mamei, M., Zambonelli, F.: Programming stigmergic coordination with the tota middleware. In: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems. ACM Press, Utrecht, Netherlands (2005) 415 – 422
13. Gelernter, D., N.Carriero: Coordination languages and their significance. Communication of the ACM **35** (1992) 96 – 107
14. Butler, Z., Kotay, K., Rus, D., Tomita, K.: Generic decentralized locomotion control for lattice-based self-reconfigurable robots. International Journal of Robotics Research **23** (2004) 919 – 938
15. Nagpal, R., Kondacs, A., Chang, C.: Programming methodology for biologically-inspired self-assembling systems. In: Proceedings of the Spring Symposium on Computational Synthesis. AAAI Press, Stanford, California, USA (2003)
16. Yim, M., Zhang, Y., Duff, D.: Modular robots. IEEE Spectrum (2002)
17. Wolpert, D., Wheeler, K.R., Tumer, K.: General principles of learning-based multi-agent systems. In: Proceedings of the International Conference on Autonomous Agents. ACM Press, Seattle, Washington, USA (1999)
18. Wolfram, S.: A New Kind Of Science. Wolfram Media (2002)
19. Mamei, M., Roli, A., Zambonelli, F.: Emergence and control of macro spatial structures in perturbed cellular automata, and implications for pervasive computing systems. IEEE Transactions on Systems, Man, and Cybernetics **35** (2005) 337 – 348